

Fig. 1

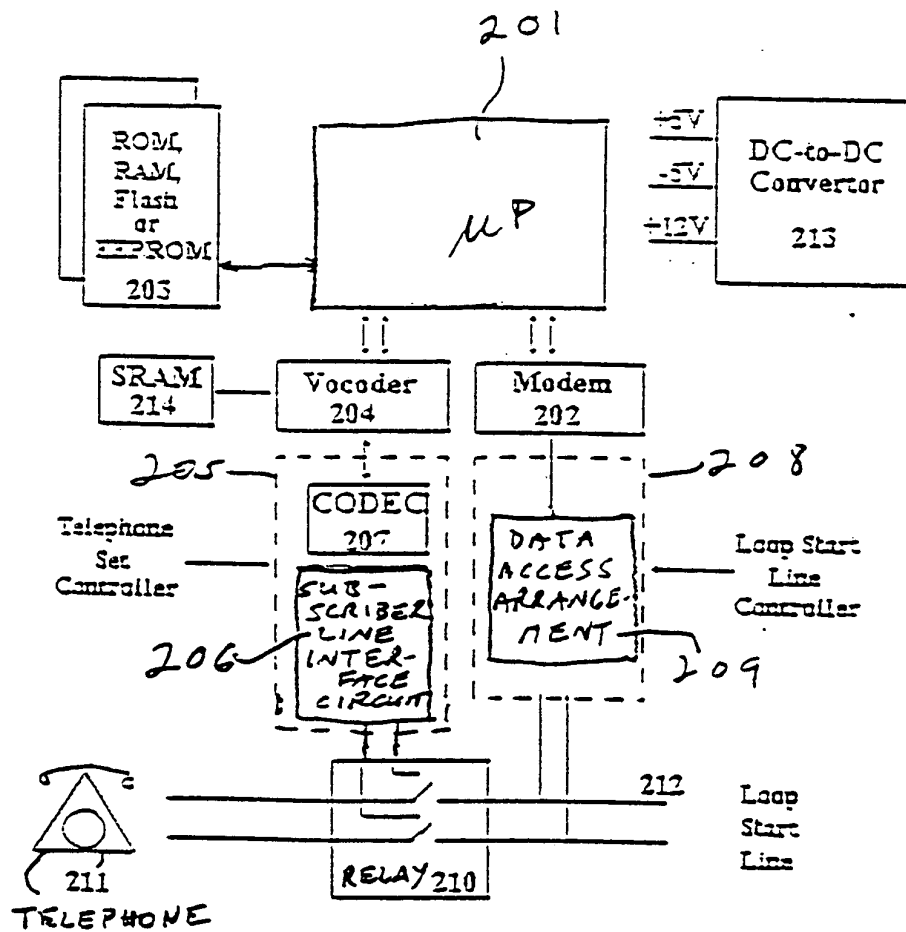


Figure 2

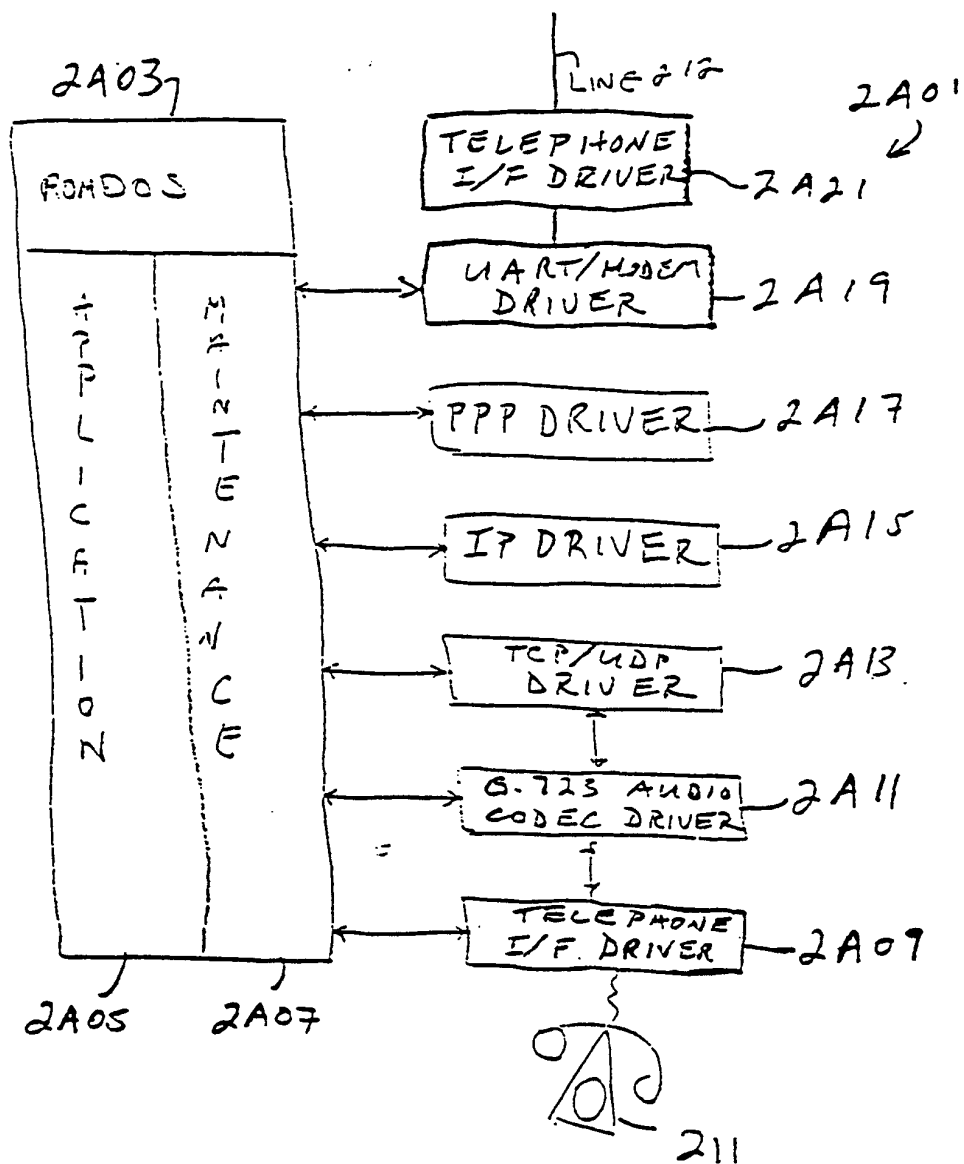
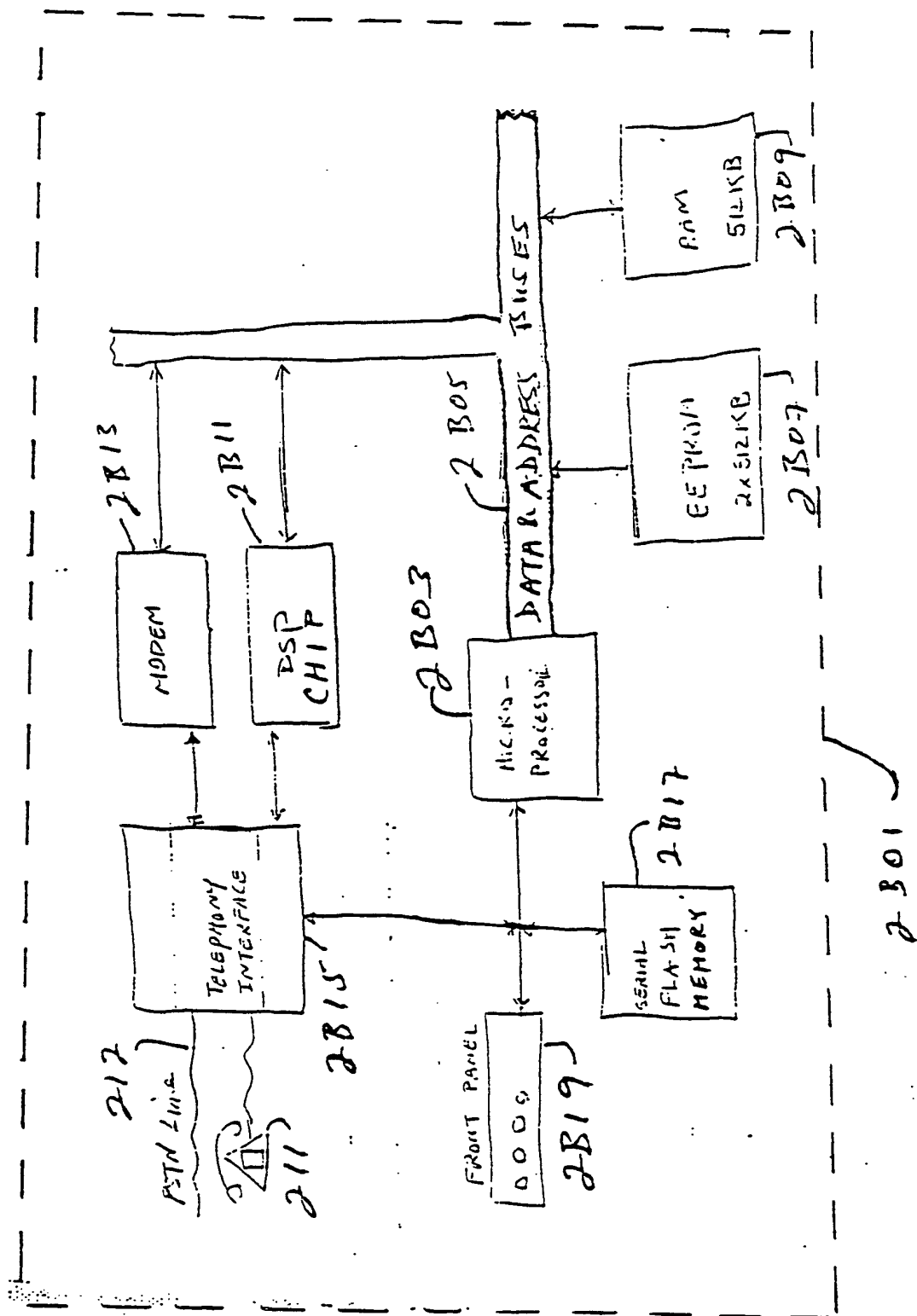


Fig. 2A

Fig. 2B



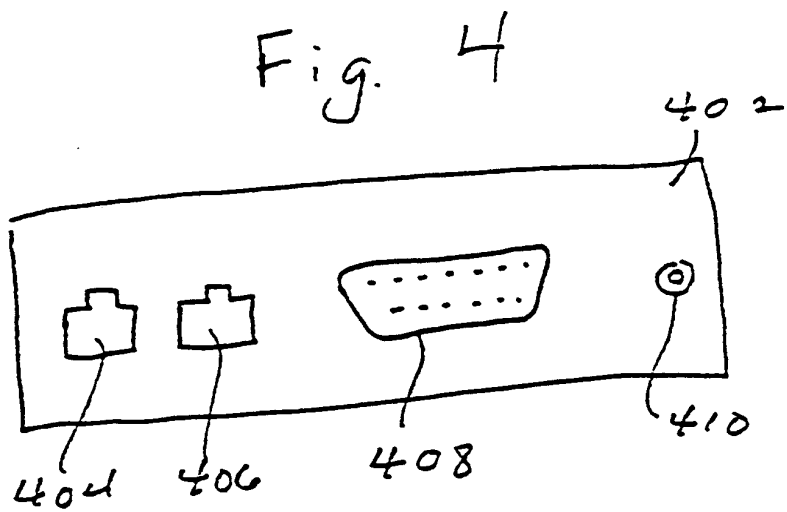
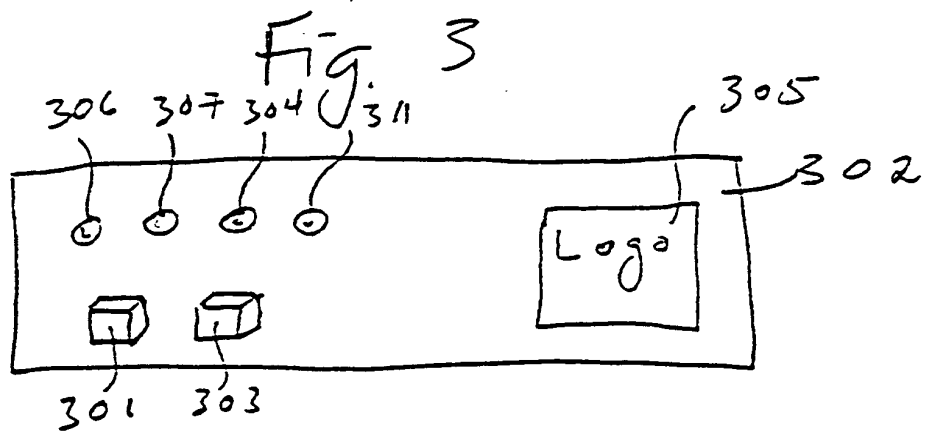


Fig. 5

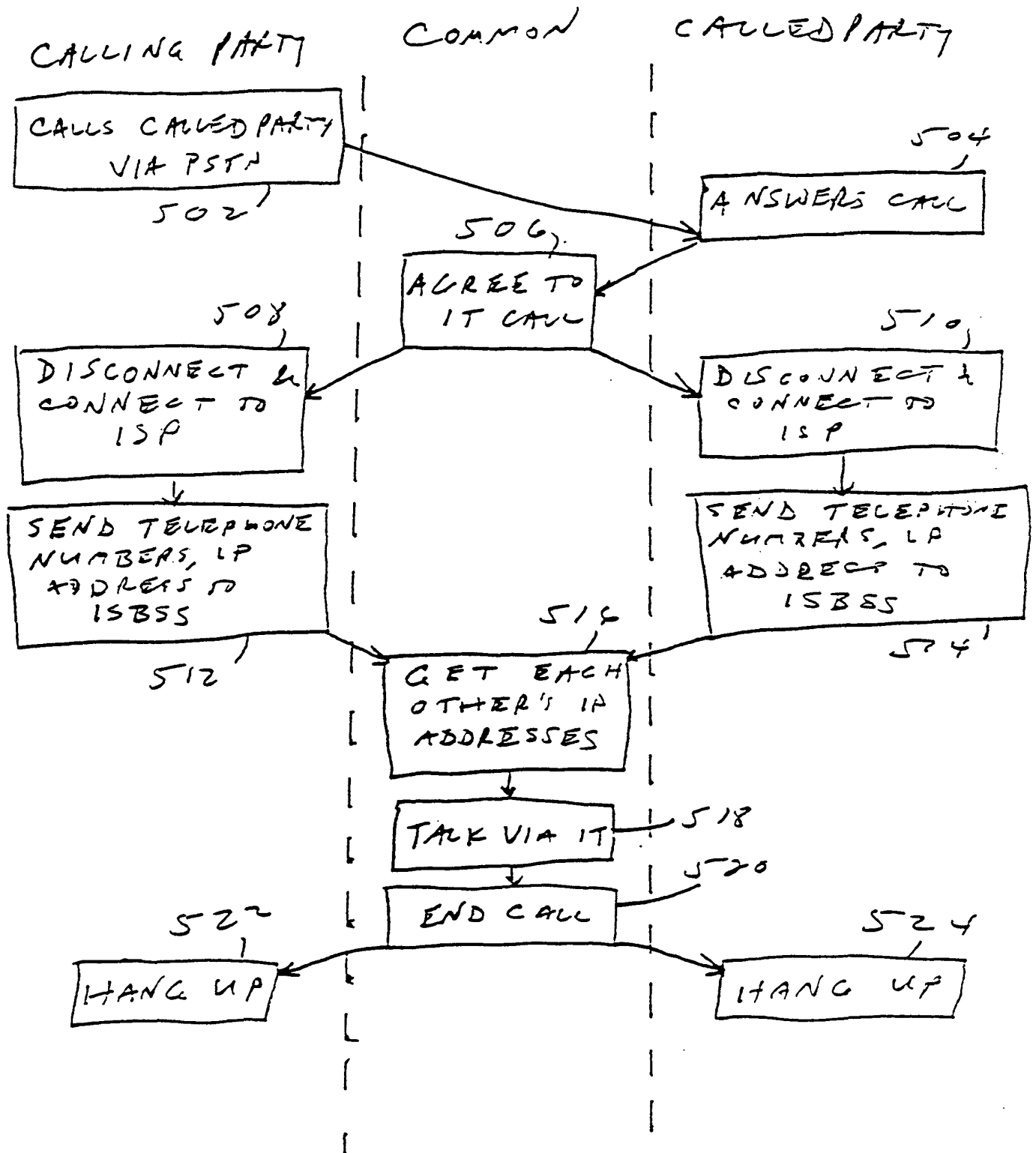


Fig. 6

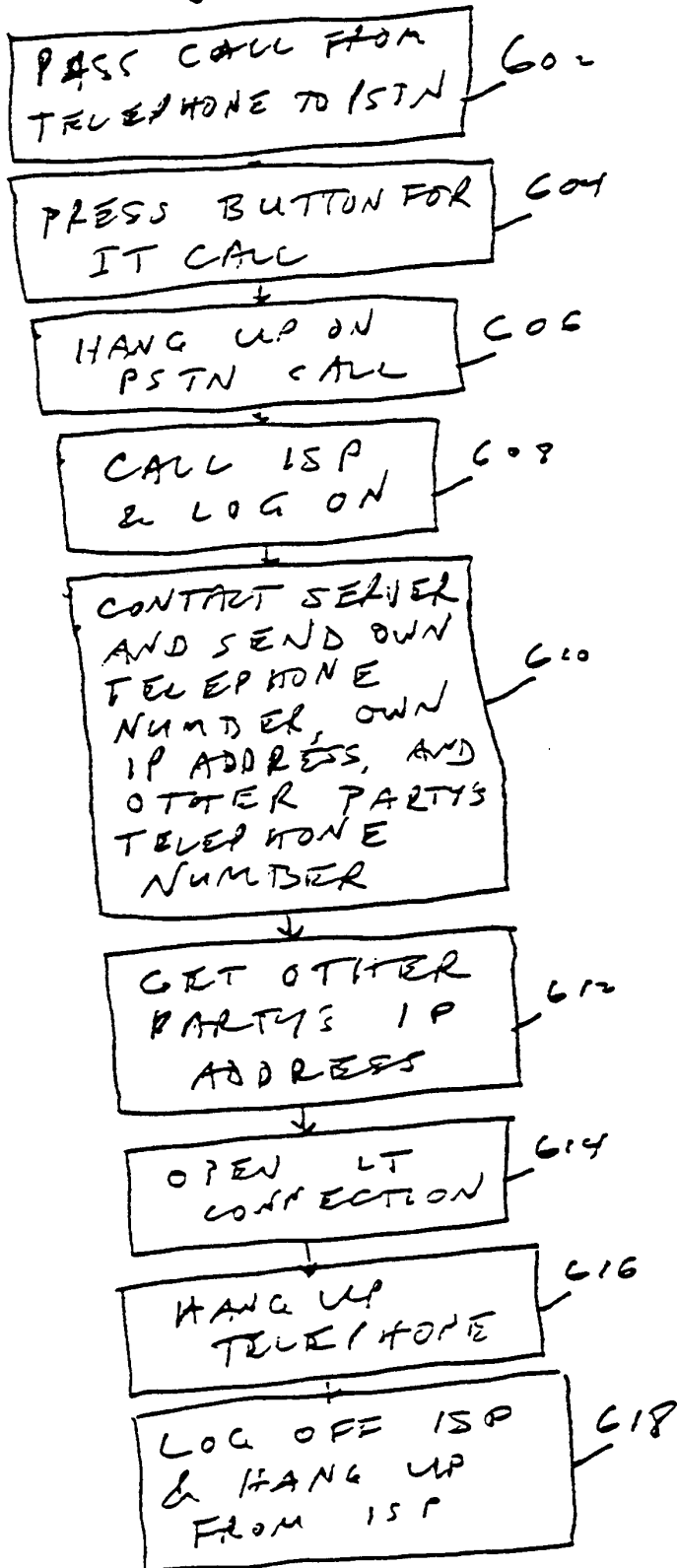


Fig. 7A

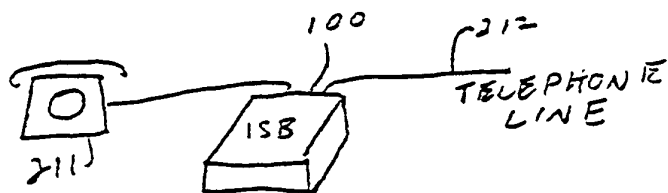


Fig. 7B

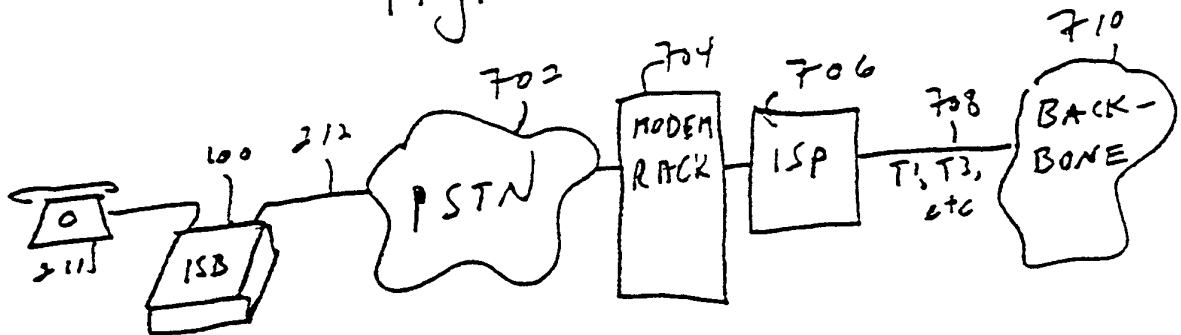
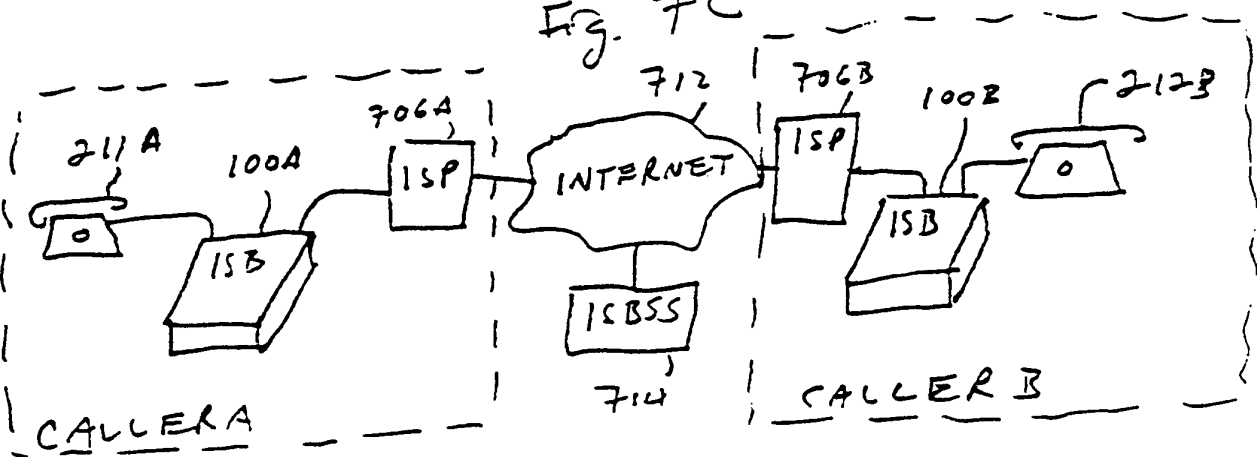


Fig. 7C



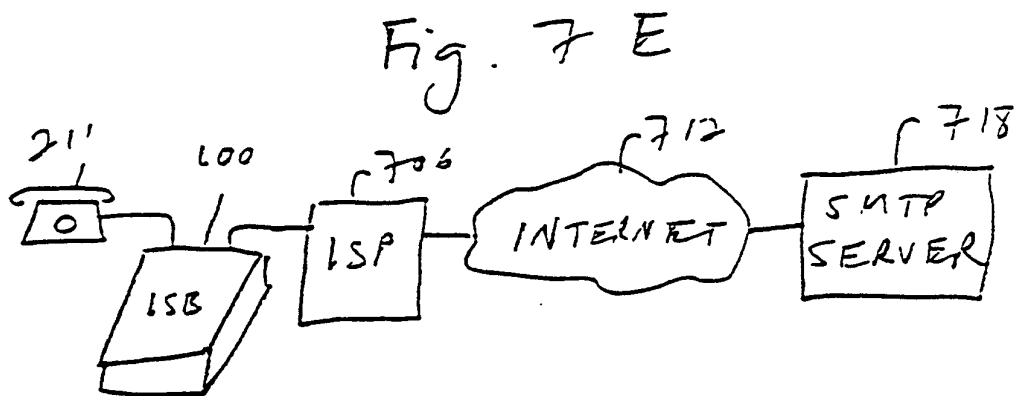
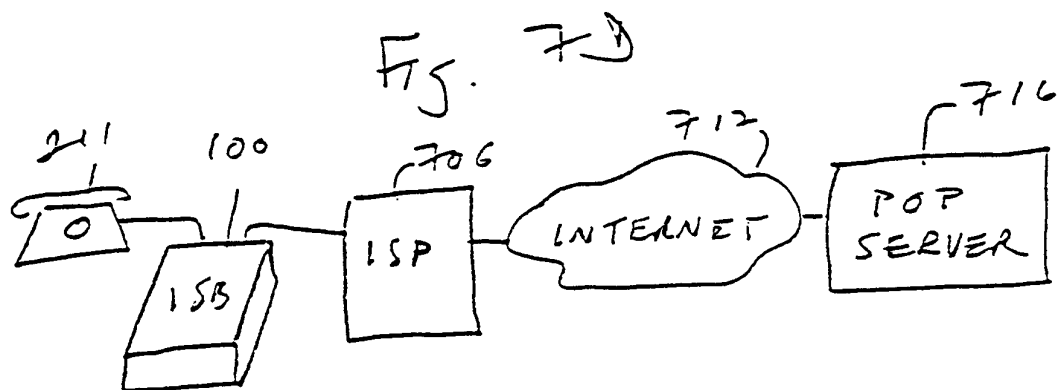


Fig. 8

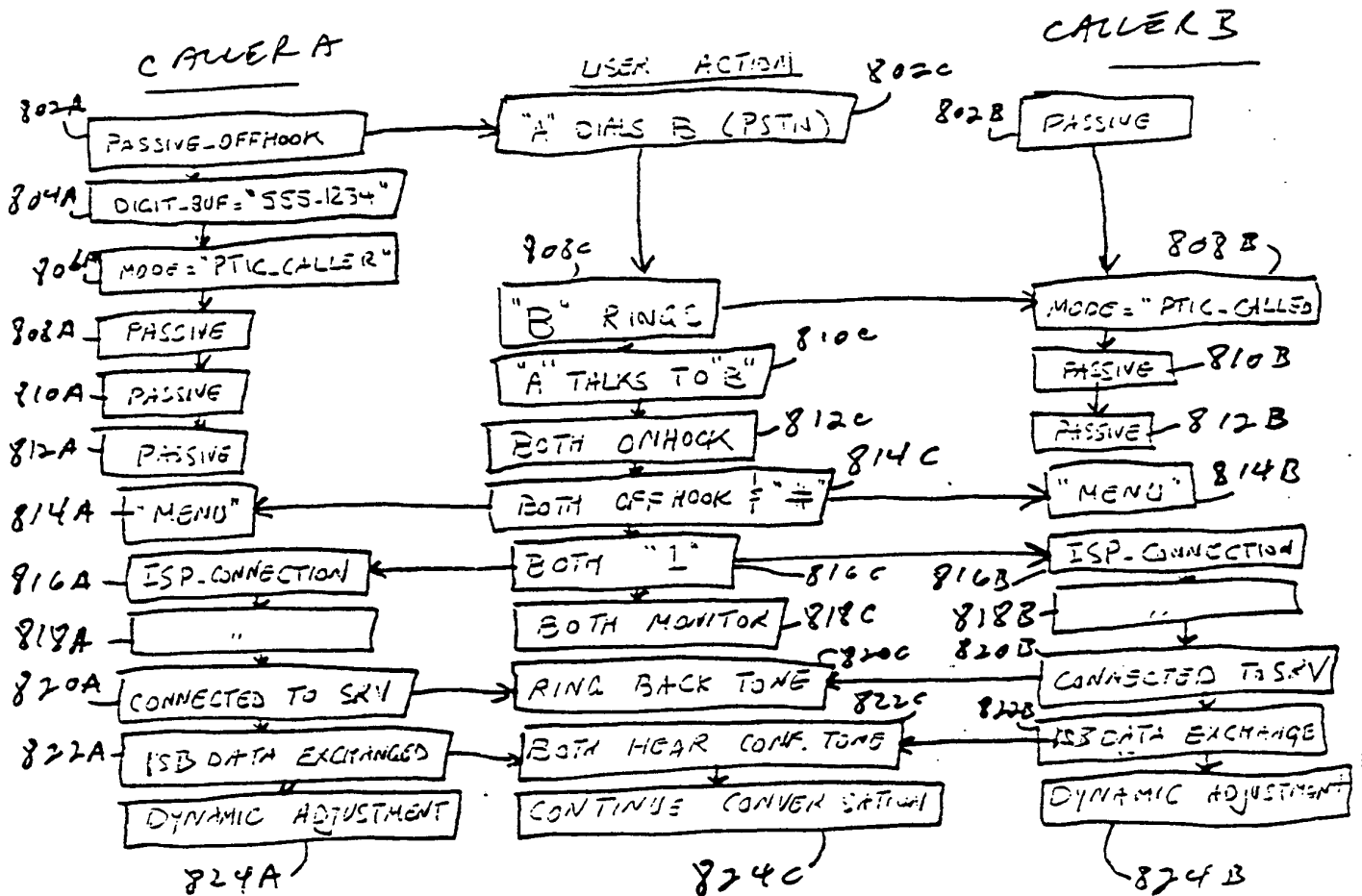
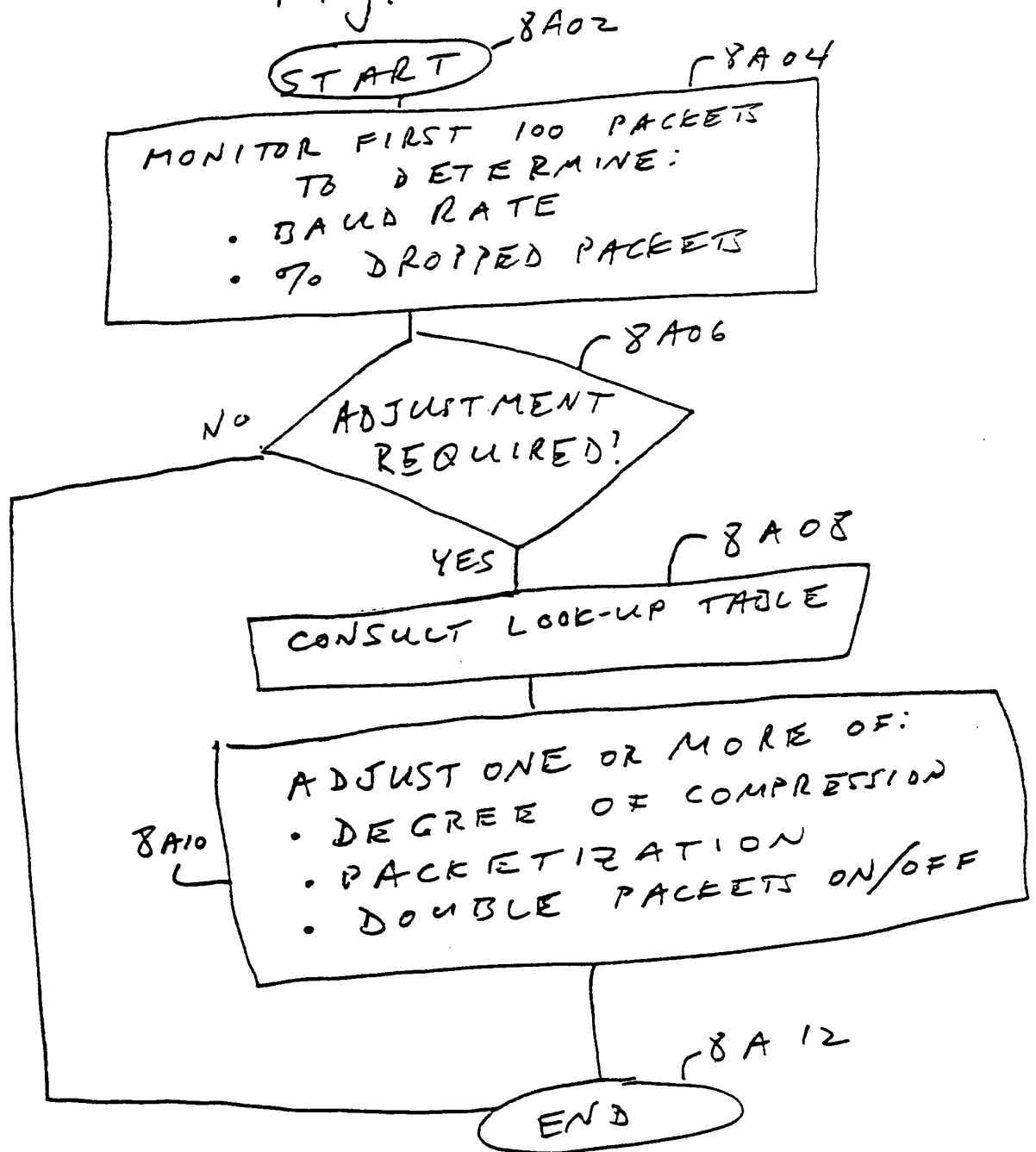


Fig. 8A



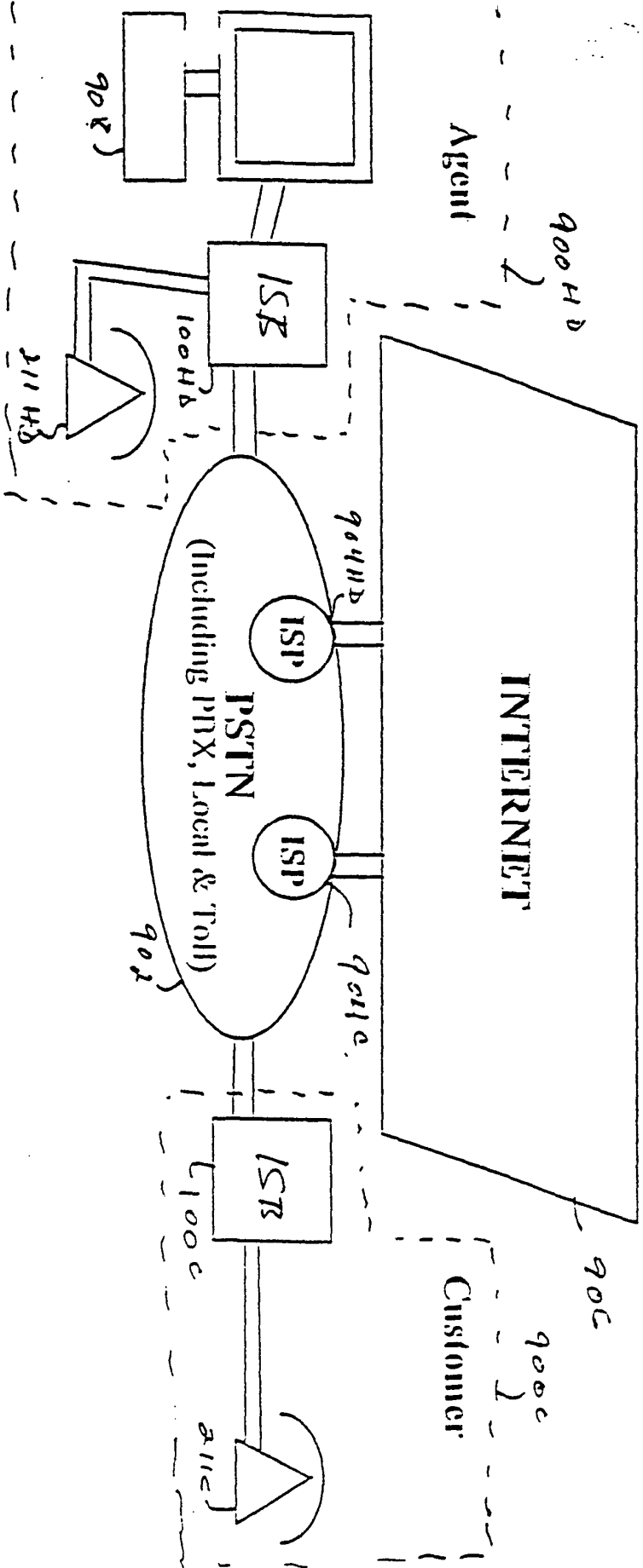


Fig. 9

Fig. 9A

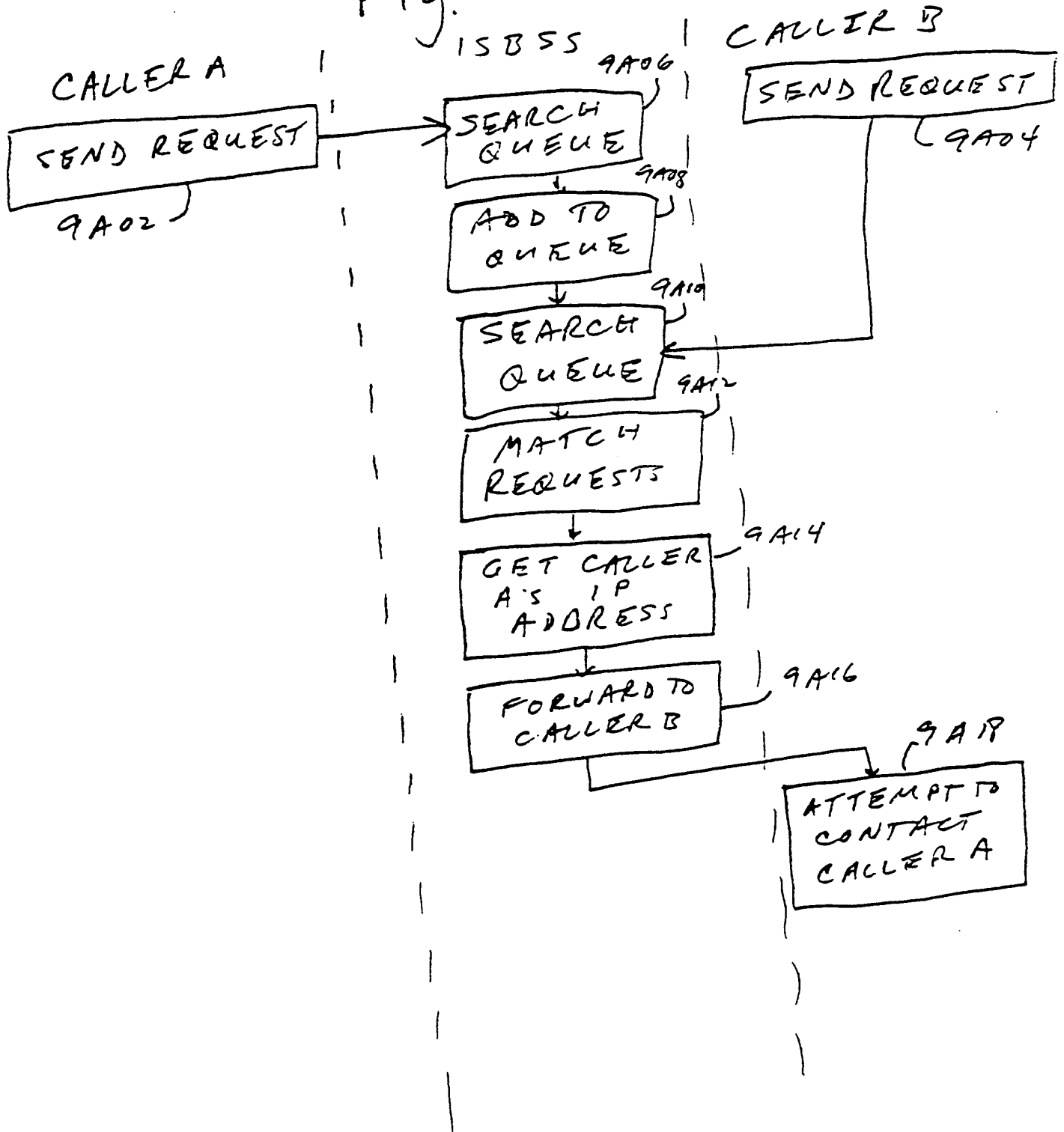


Fig. 10A

```
unsigned char *itobcd (unsigned int decimalNumber, unsigned int *digitCount)
/*
```

given an integer, produces and returns a BCD (binary-coded decimal) string, in which each byte (unsigned char) is logically split into two 4-bit "nibbles", each of which contains one digit of the original integer. Also returned as an argument-by-reference is the number of digits found in the original integer, which is useful for later manipulations.

The most significant digit of the original integer is stored "first", i.e. in the high-order nibble of the leftmost byte of the BCD string.

In the current implementation, (non-leading) zeroes in the original integer are stored as hex digit 'A' (0xA) in order to distinguish them from "blank" or "filler" nibbles and/or bytes, which actually contain zeroes.

```
*/
```

```
{
    // these are static to reduce repeat memory allocation-- for FoneFriend

    static int numOfBytes;           // bytes needed to store it as BCD
    static int numOfDigits;          // for internal use only!
    static unsigned char *BCDbuf;    // the return value goes here
    static unsigned char *bytePtr;   // moving pointer for loading BCDbuf...
    static char BitShift;            // used for decimal-to-hex conversion
    static char BCDdigits[10] =      // this allows us to do tricks like
        { 0xA, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // storing digit 0 as 0xA

    // figure out the number of digits in 'decimalNumber'
    numOfDigits = log10((double) decimalNumber) + 1;
    if (numOfDigits <= 0)
        return NULL;
    digitCount = numOfDigits; // digitCount is returned to the user

    numOfBytes = (int) ceil((double) numOfDigits / 2.0);

    // set up storage and pointers accordingly
    BCDbuf = new unsigned char[numOfBytes];
    bytePtr = BCDbuf[numOfBytes-1];

    // clear out the contents of BCDbuf-- correct functioning depends on this
    bzero(BCDbuf, numOfBytes);
}
```

Fig. 10 B

```
// we are storing BCD digits from most to least significant, going
// left to right; and there are two digits per byte. If there are
// an odd number of digits to store, then the least significant decimal
// digit will wind up in the HIGH-order nibble of the last (rightmost)
// byte used; if there are an even number of digits, this last digit
// will end up in the LOW-order nibble of the last byte. Since we start
// by storing the least significant decimal digit and move backwards,
// we have to know right away which nibble to put it in. QED.
if (numOfDigits % 2) // we have an odd number of digits
    BitShift = 4; // start in high-order nibble (left-shift 4 bits)
else BitShift = 0; // start in low-order nibble (no shift)

while (numOfDigits-- > 0) { // we have at least one more digit to do

    // get the last digit of 'decimalNumber' and put it in the
    // appropriate nibble
    *bytePtr += (BCDdigits[decimalNumber % 10] << BitShift);

    // now, we need to get ready to deal with the next digit.
    // cruddy code alert! BitShift can have the values 0 and 4; if it
    // is currently 0, then we just handled the LOW-order nibble of a
    // byte, and we will stay within this byte to do the next digit.
    // But if BitShift is currently 4, we just did the HIGH-order byte
    // and we can move back to the previous byte. The following
    // very confusing code does that for you:
    bytePtr += (BitShift / 4);

    // of course, the value of BitShift must now be toggled:
    BitShift = 4 - BitShift;

    // finally, we line up 'decimalNumber' to deal with the next digit
    // in line, by way of throwing away the last digit we looked at, which
    // was the least significant digit of 'decimalNumber'.
    decimalNumber /= 10;

    // at long last, we're ready to copy the digit into the BCD string:
    // *bytePtr += (BCDdigits[decimalNumber % 10] << BitShift);

}

return BCDbuf;
}
```

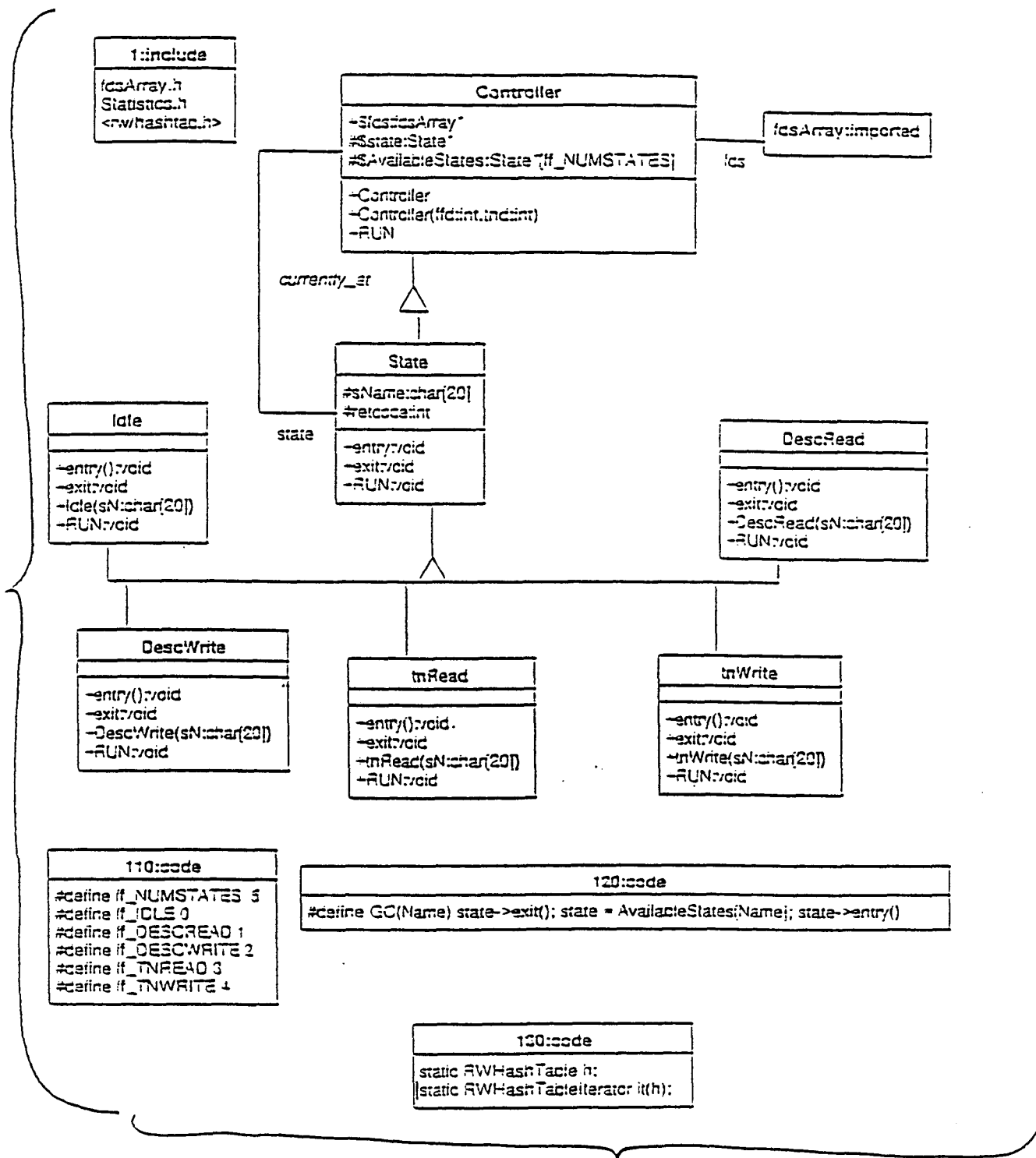


Fig. 11

Fig. 11A

```
typedef struct {
    unsigned short struct_type;
    // tells us how to interpret the tx_data
    // 1001 t_ConnectPacket
    // 1002 t_RxConnectPacket
    unsigned short len; // length of data in tx_data
    char tx_data[252]; // 252 bytes to handle future expansion
} tx_packet;
```

Fig. 11 B

```
typedef struct {
    unsigned char hv_version; // identifies the originator of this struct
    unsigned char sv_version; // 1 == 1st version
    // the connection type should be the first bytes read.
    // the types are:
    // 1 - caller non-1st time
    // 2 - called
    // 3 - caller 1st time
    // 4 - nmic
    // 5 - message
    // 7 - self-test
    // 8 - upgrade request
    unsigned short int connect_type;
    unsigned char my_phone_num[8];
    unsigned char his_phone_num[8];
    unsigned long my_serial_num;
    unsigned long his_serial_num;
    unsigned char my_ip[4];
    t_BillingData bill_rec;
} t_ConnectPacket;
```

Fig. 11 C

```
typedef struct {
    unsigned long start_time; // start time of previous service
    unsigned long stop_time; // duration (in seconds) of previous service
    unsigned char phone[8]; // phone number of previous call
    unsigned char stat_data[8]; // statistical data about previous service
} t_BillingData;
```

Fig. 11 D

```
typedef struct {
    unsigned short struct_type;
        // tells us how to interpret the tx_data
        // 1001 t_ConnectPacket
        // 1002 t_RxConnectPacket
    unsigned short len; // length of data in tx_data
    char tx_data[252]; // 252 bytes to handle future expansion
} tx_packet;
```

Fig. 11 E

```
typedef struct {
    // New fields added to allow for commands
    unsigned char pkt_type; // 0 == message, 1 == error
    unsigned char me_type;
        // messages:
        // 0 = return usable IP addr,
        // 1 = no match; IP == 0.0.0.0,
        // 2 = go to another server; IP address given
        // 3 = no action to take (response to message or self-test; IP == 0.0.0.0)
        // errors:
        // 0 = problem on my end; retry from scratch
        // 1 = problem with your data; retry from scratch
        // 2 = you are not an active user of the requested FF Service.
    unsigned char commandType;
        // 0 == no command
        // 1 == contact command server for further commands
        //      send new IP addr in command
        // 2 == set Update Available light on
        // 3 == unset Update Available light
        // 4 == new main server
        //      send new IP addr in command
        // 5 == new backup server
        //      send new IP addr in command
    unsigned char commandSize; // number of bytes found in command[]
    unsigned char his_ip[4];
    unsigned long cur_time;
    char command[32];
        // If commandSize <= 28 we can rely on
        // bytes command[28] .. command[31] containing the
        // sender serial number just for debugging purposes.
        // we have not specified what a command looks like.
        // commandType == 2:
        //      commandSize = 8, command = "10 2 1\r\n"
        // commandType == 4:
        //      commandSize = 21, command = "0 1 0 137 140 7 222\r\n"
        // commandType == 21:
        //      commandSize = 8, command = "0 1 1 137 140 7 222\r\n"
} t_RxConnectPacket;
```

Fig. 11 F

***** Results from generation of Statistics *****

***** Absolute Value Counters *****

```

n Entered Idle state      : 985131
n FFServer connection Requests: 0
n Entered DescRead state  : 0
n Entered DescWrite state : 0
n DescRead ok             : 0
n DescRead failed: wrong size : 0
n DescRead failed: disconnect : 0
n DescRead failed: orderly rel: 0
n DescWrite ok            : 0
n DescWrite failed        : 0
n Init New Descriptor     : 1
n Conn discon in complete list: 0
n Invalid Client Port     : 0
n Entered Housekeeping    : 985099
n Completed Connection RQ : 0
n Expired Connection RQ   : 0
n Inactive Connection RQ  : 0
n tnClient Write ok       : 29
n tnClient Write failed   : 0
n Serial Number Invalid   : 0

```

***** Maximum Value Counters *****

```

n Max Complete Connection Q : 0
n Max Stack Size            : 0
n Max Connection List Size   : 0

```

***** Minimum Value Counters *****

```

n Min Stack Size           : 2147483647
n Min Connection List Size : 0

```

***** End of StatisticsReport *****

Monitoring Stopped

Fig. 11 G

Mon Feb 23 13:06:31 1998> New logged session of FFServer

Mon Feb 23 13:06:31 1998> Number of Invalid Serial Numbers: 1000

Mon Feb 23 13:06:55 1998> New TNClient (IP.Port): 137.140.8.104.36239

Mon Feb 23 13:07:56 1998> Closing TNClient (IP.Port): 137.140.8.104.36239

Mon Feb 23 13:07:56 1998> (CL) Unknown ConnectType (IP.Port): 137.140.8.104.36239

Mon Feb 23 13:07:57 1998> (CL) Wrong Packet Size (IP.Port): 137.140.8.104.36239

Mon Feb 23 13:07:58 1998> (CL) PcktType != 1001 (IP.Port): 137.140.8.104.36239

Mon Feb 23 13:07:59 1998> (CL) tn_packetPtr was NULL (IP.Port): 137.140.8.104.36239

Mon Feb 23 13:07:59 1998> (CL) Failed on attempt to insert (IP.Port): 137.140.8.104.36239

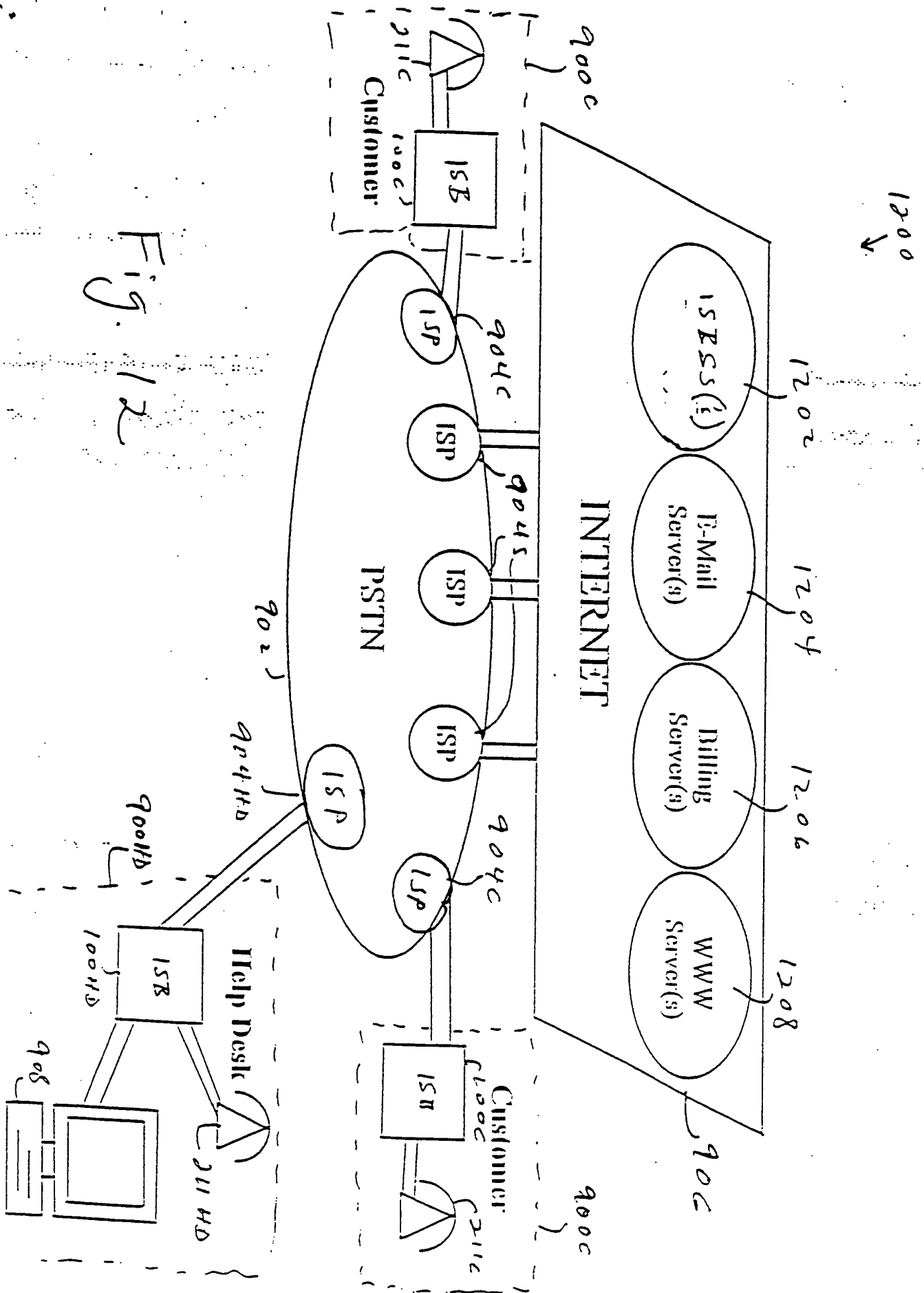


Fig. 12

Fig. 13

1300
↓

